

Les processus : fork, exit et wait

Luigi Santocanale

Laboratoire d'Informatique Fondamentale,
Centre de Mathématiques et Informatique,
39, rue Joliot-Curie - F-13453 Marseille

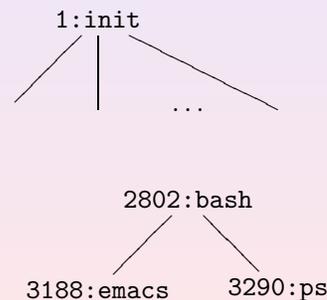
3 octobre 2005

Plan

- 1 Le processus : vue utilisateur
- 2 Les processus : principes d'implémentation
 - Le bloc de contrôle
 - Le contexte d'un processus
- 3 Appels systèmes POSIX
 - Création d'un processus : la primitive `fork`
 - Terminaison d'un processus : `exit`
 - Terminaison et synchronisation : `wait`
 - Accès aux données du bloc de contrôle du processus

Arborescence des processus

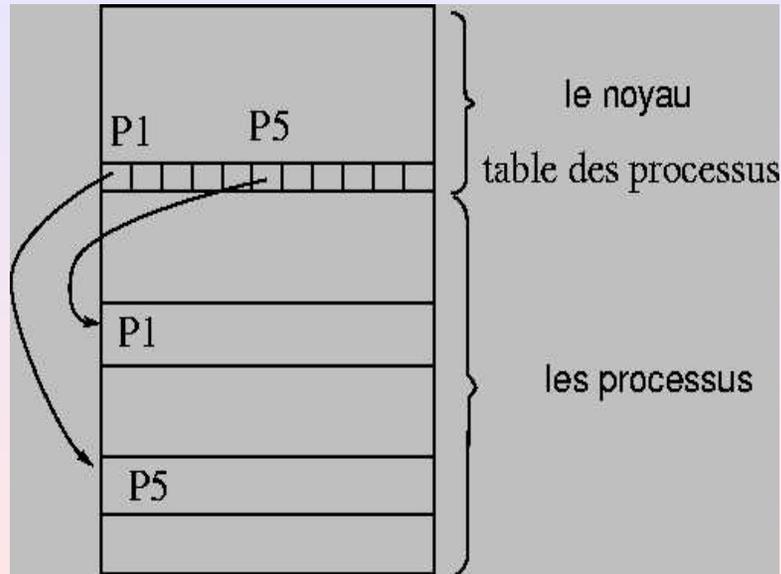
```
[lsantoca@localhost lecture3]$ ps -o "%p %P %c"
PID  PPID  COMMAND
2802  2799  bash
3188  2802  emacs
3290  2802  ps
```



Implémentation des processus

Un processus comporte:

- un espace d'adressage,
- le bloc de contrôle du processus, décomposé en :
 - entrée dans la table des processus,
 - une zone `u`, alloué dynamiquement à la création du processus.
- son contexte.



La table des processus

Tableau de structures proc, voir sys/proc.h :

S (état)
adresse zone u
adresses en mémoire
PID (ID du processus), PPID (ID du père)
événement en attente (si endormi)
NI (valeur nice), priorités
ensemble des signaux pendants
temporisation,
:

La zone u

Structure user, voir sys/user.h :

pointeur sur l'entrée dans la table des processus
RUSER, EUSER, RGROUP, EGROUP
consommation CPU
masque des signaux
terminal de contrôle
errno
. et /
table des descripteurs ouverts
masque création
:

Accès au bloc de contrôle par la commande ps

```
[lsantoca@localhost lecture3]$ ps -eo cmd,pid,ppid,ruser,euser,\
> rgroup,egroup,s,ni | grep ps
cupsd                2053      1 root      root      root      root      S    0
cups-pollD 193.5     2244    2053 lp        lp        sys       sys       S    0
cups-pollD 193.5     2245    2053 lp        lp        sys       sys       S    0
cups-pollD 192.1     2246    2053 lp        lp        sys       sys       S    0
kdeinit: dcopter     3254      1 lsantoca lsantoca lsantoca lsantoca S    0
man ps               3664    3317 lsantoca lsantoca lsantoca lsantoca S    0
ps -eo cmd,pid,p     3697    3329 lsantoca lsantoca lsantoca lsantoca R    0
```

Contexte d'un processus

Ensemble des données qui permettent de reprendre l'exécution d'un processus interrompu.

- mot d'état, contexte de l'unité centrale :
 - accumulateur,
 - registre d'instruction et compteur d'instruction,
 - registres d'état du processeur,
 - registres d'états du processus.
- état du processus,
- variables globales statiques dynamiques,
- entrée dans la table du processus,
- zone u,
- piles user et system,
- zones de codes et de données.



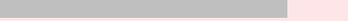
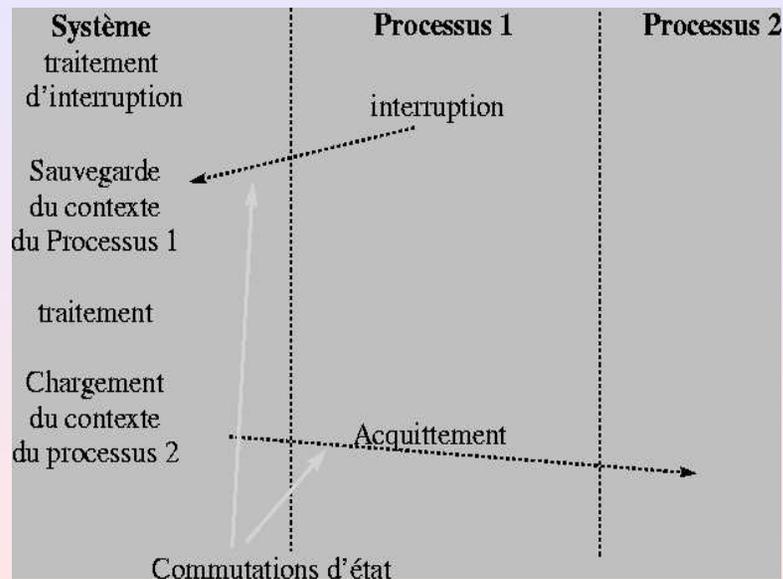
Commutation du mot d'état

Il s'avère à la réception/acquittement d'une interruption.
Trois types d'interruptions :

- externe : horloge, disque, console, périphérique.,
- déroutement : erreur du processeur, division par zéro, page fault,
- appel système.



Traitement des interruptions



fork

```
#include <unistd.h>  
pid_t fork(void );
```

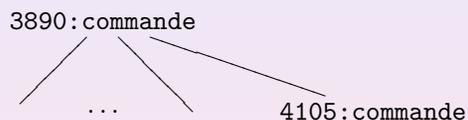
Retourne :

- -1, si erreur,
- 0, au fils,
- pid du fils, retourné au père.

Sommaire : Clonation du processus



Naissance des processus



Détails

Dans un appel à la primitive fork :

- TOUS les données concernant le processus sont dupliquées :
 - espace d'adressage,
 - bloc de contrôle du processus.
- À L'EXCEPTION de :
 - la région texte peut être partagée entre les deux processus,
 - le PID et le PPID sont mis à jour.

Algorithme fork

(Bach 1989)

```
algorithme fork
entrée: néant
sortie: au processus parent, le PID du fils
        au processus fils, 0
(
  vérifier que des ressources noyau sont disponibles;
  accéder à un emplacement vide de la table des processus
                                et à un PID unique;
  vérifier que l'utilisateur n'a pas trop de processus en exécution;
  marquer l'état du fils "en création";
  copier les données de l'emplacement du parent dans la table
                                des processus dans celui du fils;
  incrémenter les comptes i-noeuds du répertoire courant et
                                du répertoire racine (si changé);
  incrémenter les comptes des fichiers ouverts
                                dans la table des fichiers;
  faire une copie du contexte du parent (zone u, code, données, pile)
                                en mémoire;
  empiler la couche contexte factice du niveau système dans le
                                contexte du niveau système du fils;
                                le contexte factice contient des données qui
                                permettent au processus fils de se reconnaître et d'être
                                lancé au moment où il est élu en vue de son exécution;
  if (processus en exécution est le processus parent)
  (
    changer l'état du fils à celui de "prêt à l'exécution";
    return (ID du fils); /* du système à l'utilisateur */
  )
  else /* le processus en exécution est le processus fils */
  (
    initialiser les champs temps de la zone u;
    return (0); /* à l'utilisateur */
  )
)
```

fork

```
#include <unistd.h>
pid_t fork(void );
```

Retourne :

- -1, si erreur,
- 0, au fils,
- pid du fils, retourné au père.

Sommaire : Clonation du processus

Programme : exfork.c

```
1 : #include <stdio.h>
2 : #include <unistd.h>
3 : #include <sys/types.h>
4 :
5 : int main(void)
6 : {
7 :     pid_t pid;
8 :
9 :     switch (pid = fork())
10 :    {
11 :    case -1:
12 :        perror("fork");
13 :        return -1;
14 :    case 0:
15 :        printf("Je suis le fils et mon pid est %d.\n", getpid());
16 :        return 0;
17 :    default:
18 :        printf("Je suis le père, "
19 :              "mon pid est %d, j'ai un fils avec pid %d.\n", getpid(),
20 :              pid);
21 :        return 0;
22 :    }
23 : }
```

Programme : exfork2.c

```
1 : #include <stdio.h>
2 : #include <stdlib.h>
3 : #include <unistd.h>
4 : #include <sys/types.h>
5 :
6 : int main(void)
7 : {
8 :     pid_t pid;
9 :
10 :    if ((pid = fork()) == -1)
11 :    {
12 :        perror("fork");          /* Erreur */
13 :        exit(EXIT_FAILURE);
14 :    }
15 :
16 :    if (pid == 0)                 /* Si fils */
17 :        code_du_fils();
18 :    else                          /* Si père */
19 :        code_du_pere();
20 : }
```

Algorithme exit

(Bach 1989)

```
algorithme exit
entrée: code de retour pour le processus parent
sortie: néant
(
    ignorer tous les signaux;
    if (moniteur du groupe de processus associé au terminal de contrôle)
    (
        envoyer un signal "coupure de ligne" à tous les membres
        du groupe de processus;
        mettre à 0 le groupe de processus de tous les membres;
    )
    fermer tous les fichiers ouverts
    (version interne de l'algorithme close);
    libérer le répertoire courant (algorithme iput);
    libérer le répertoire racine (changé) courant, s'il existe
    (algorithme iput);
    libérer les régions, la mémoire associée au processus
    (algorithme freereg);
    écrire un enregistrement de statistiques;
    rendre l'état du processus zombie;
    affecter l'ID processus du parent de tous les processus fils au
    processus init (1);
    si un des fils est zombie, envoyer le signal "mort d'un fils" à init;
    envoyer le signal "mort d'un fils" au processus parent;
    changement de contexte;
)
```

exit, atexit, abort, _exit

```
#include <stdlib.h>
#include <unistd.h>
```

```
void _exit(int status);
```

Sommaire : Fermeture des fichiers ouverts, terminaison normale avec code *status* : ce paramètre est reçu par le père à l'aide de la primitive *wait*.

```
void exit(int status);
```

Sommaire : Exécute les fonctions enregistrées par *atexit*, vide les tampons de la bibliothèque standard, ferme les fichiers ouverts avec *fopen*, appel à la primitive *_exit* avec paramètre *status*.

```
int atexit(void (*valeur)(void));
```

```
void abort(void);
```

Sommaire : Vide les tampons de la bibliothèque standard, terminaison anormale.

Programme : exexit.c

```
1 : #include <stdlib.h>
2 : #include <stdio.h>
3 : #include <unistd.h>
4 :
5 : void terminer(void)
6 : {
7 :     printf("\nAppel à la fonction terminer ");
8 : }
9 :
10 : int main(int argc, char *argv[])
11 : {
12 :
13 :     atexit(terminer);
14 :
15 :     printf("\nUtilisation de \"%s\" pour sortir :", argv[1]);
16 :     printf("\nContenu du tampon à vider ");
17 :     SORTIE;
18 : }
```

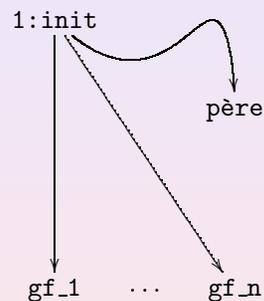
Session : exit

```
[lsantoca@localhost lecture4]$ for i in "exit(0)" \
> "abort()" "_exit(0)";\
> do gcc -Wall -pedantic -DSORTIE=$i exexit.c;\
> a.out $i; done
```

Utilisation de "exit(0)" pour sortir :
Contenu du tampon à vider
Appel à la fonction terminer
Utilisation de "abort()" pour sortir :
Contenu du tampon à vider Aborted

Utilisation de "_exit(0)" pour sortir :

Mort des processus, synchronisation



Algorithme wait

(Bach 1989)

```
algorithme wait
entrée: adresse d'une variable pour ranger l'état du
        processus qui fait l'exit.
sortie: ID et code d'exit du fils
(
  if (le processus en attente n'a pas de processus fils)
    return (erreur);
  for(;;) /* boucle jusqu'à un return */
  (
    if (le processus en attente a des fils zombies)
    (
      accéder à un fils zombie quelconque;
      rajouter l'utilisation de l'UC par le fils au parent;
      libérer l'élément de la table des processus
      correspondant au fils;
      return (ID et code d'exit du fils);
    )
    if (le processus n'a pas de fils)
      return (erreur);
    s'endormir en une priorité interruptible en attente de
    l'événement: exit du processus fils;
  )
)
```

wait, waitpid

```
#include <sys/types.h>
#include <sys/wait.h.h>
```

```
pid_t wait(int * ptr);
    ptr : un adresse à remplir avec le code renvoyé par le fils.
Retourne : pid du fils, où -1 si échec.
```

```
pid_t waitpid(pid_t pid, int * ptr, int opts);
    pid : > 0 : un processus
         0 : tous les processus du groupe
        -1 : tous les fils
        <-1 : tous les processus dans le groupe |pid|
```

```
    ptr : adresse à remplir
    opts : |de :
           WNOHANG : sans être bloqué
           WUNTRACED : si le processus est stoppé
```

Retourne : pid du processus pris en compte, -1 erreur, 0 échec
avec option WNOHANG

Programme : exwait.c

```
1 : #include <wait.h>
2 : #include <stdio.h>
3 : #include <unistd.h>
4 : #include <stdlib.h>
5 : #include <time.h>
6 :
7 : void fils(void);
8 : void pere(pid_t pid);
9 :
10 : int main(void)
11 : {
12 :     pid_t pid;
13 :     srand(time(NULL));
14 :
15 :     switch (pid = fork())
16 :     {
17 :     case -1: /* Erreur */
18 :         exit(EXIT_FAILURE);
19 :     case 0:
20 :         fils();
21 :     default:
22 :         pere(pid);
23 :     }
24 :     exit(EXIT_SUCCESS);
25 : }
26 :
```

Décodage de la valeur de sortie du fils

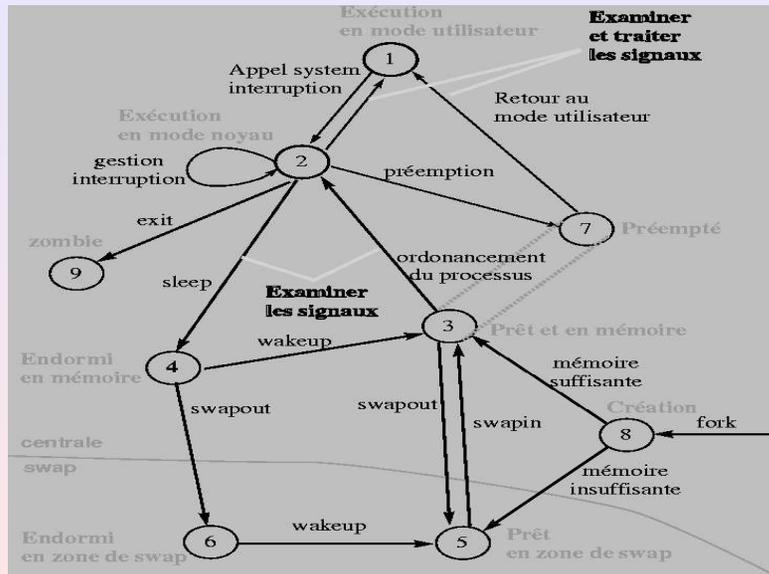
Macros définies dans wait.h :

- WIFEXITED(code) :
vrai si le processus s'est terminé de façon normale
 - WEXITSTATUS(code) :
le code de sortie, c.à.d. *status* dans *exit(status)*.
- WIFSIGNALED(code):
vrai si le processus s'est terminé à cause d'un signal
(par exemple, un CTRL-C)
 - WTERMSIG(code), le signal qui a provoqué la terminaison.
- WIFSTOPPED(code) :
vrai si le processus est arrêté à cause d'un signal
(primitive *waitpid* avec option *WUNTRACED*)
 - WSTOPSIG(code), le signal qui a provoqué l'arrêt.

Programme : exwait.c

```
27 : void fils(void)
28 : {
29 :     int signum = rand() % (NSIG - 1) + 1;
30 :     printf("Signal choisi : %d.\n", signum);
31 :     kill(getpid(), signum);
32 :     exit(signum);
33 : }
34 :
35 : void pere(pid_t pid1)
36 : {
37 :     int codesortie;
38 :     pid_t pid2 = waitpid(pid1, &codesortie, WUNTRACED);
39 :
40 :     if (WIFSIGNALED(codesortie))
41 :         printf("Le fils %d s'est terminé à cause du signal %d.\n", pid2,
42 :             WTERMSIG(codesortie));
43 :     if (WIFEXITED(codesortie))
44 :         printf("Le fils %d s'est terminé normalement"
45 :             " avec code sortie %d.\n", pid2, WEXITSTATUS(codesortie));
46 :     if (WIFSTOPPED(codesortie))
47 :         printf("Le fils %d est stoppé" " avec code %d.\n", pid2,
48 :             WSTOPSIG(codesortie));
49 :     exit(EXIT_SUCCESS);
50 : }
```

États d'un processus : deuxième approximation (Revuz 1998)



Le processus : vue utilisateur
Les processus : principes d'implémentation
Appels systèmes POSIX

Création d'un processus : la primitive `fork`
Terminaison d'un processus : `exit`
Terminaison et synchronisation : `wait`
Accès aux données du bloc de contrôle du processus

ID du processus, ID du processus père

```
#include <unistd.h>  
#include <sys/types.h>
```

```
pid_t getpid(void );  
Retourne : Le pid du processus, toujours succès.
```

```
pid_t getppid(void );  
Retourne : Le pid du père, toujours succès.
```

Luigi Santocanale

Le processus

36

Le processus : vue utilisateur
Les processus : principes d'implémentation
Appels systèmes POSIX

Création d'un processus : la primitive `fork`
Terminaison d'un processus : `exit`
Terminaison et synchronisation : `wait`
Accès aux données du bloc de contrôle du processus

IDs utilisateur

```
#include <unistd.h>
```

```
uid_t getuid(void );
```

Remarques : propriétaire réel :
utilisateur qui exécute le fichier exécutable.

```
uid_t geteuid(void );
```

Retourne : Le uid, toujours succès.

Remarques : propriétaire effectif :
utilisateur qui possède l'exécutable.

```
int setuid(uid_t uit);
```

Retourne : 0/-1.

Remarques : il faut voir permissions de root.

Luigi Santocanale

Le processus

37

Le processus : vue utilisateur
Les processus : principes d'implémentation
Appels systèmes POSIX

Création d'un processus : la primitive `fork`
Terminaison d'un processus : `exit`
Terminaison et synchronisation : `wait`
Accès aux données du bloc de contrôle du processus

IDs groupe

```
#include <unistd.h>
```

```
uid_t getgid(void );
```

```
uid_t getegid(void );
```

```
int setgid(uid_t uit);
```

Remarques : Comme pour `getuid` ...

Luigi Santocanale

Le processus

38

Répertoire d'un processus, masque de création

```
#include <unistd.h>
```

```
int chdir(const char * ref);
```

Retourne : 0/-1

```
char const * getcwd(char * buf, size_t taille);
```

buf : un tampon de taille *taille*.

Retourne : *buf* succès, NULL échec

```
mode_t umask(mode_t masque);
```

masque : les permission qu'on aimerait enlever par défaut aux appels de `creat`, `open`, `mkdir`, `mkfifo`.

Retourne : La vielle masque de création

